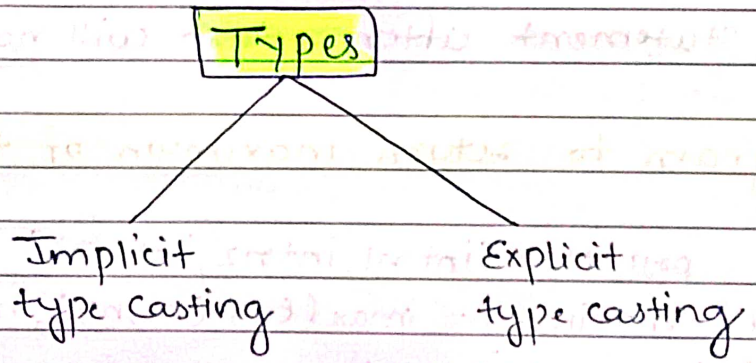
06/09/2023
Wednesday

* TYPE CASTING →

- Allows us to change the data type of a variable from one type to another.
- Crucial when we need to perform operations involving variables of different data types, ensuring that the data is handled correctly.

for instance, int ⟷ char
float ⟷ int
double ⟷ int etc.

char + int = ?? Here, type casting comes into use.



= Implicit type casting →

- Automatic type conversion.
- Compiler automatically converts one data type to another during an operation.
- This happens when we perform operations involving variables of different data types and the compiler promotes one type to a larger type to maintain precision or accuracy.

For example1,

```
int num1 = 10;
float num2 = 5.5;
float result = num1 + num2
cout << result << endl; → output = 15.5
```

Here, the int is converted into float i.e. 10.00 by compiler and float + float = float and is also stored in float data type.

example 2,

```
int num1 = 10;
float num2 = 5.5;
int result = num1 + num2;
cout << result << endl; → output = 15
```

Here, the int is converted into float by compiler and float + float = float which is converted into int as it is ~~com~~ stored in int data type.

Example 3,

```
int a = 97;
char ch = a + 1;
cout << ch << endl; → output = b
```

Example 4,

```
Char ch = 'B';
int a = ch;
cout << a << endl; → output = 66
```

__/__/__

Explicit Type Conversion →

- Manual type conversion.
- Allows us to explicitly specify the desired data type during an assignment or operation.
- You can use the casting operator, which is represented by parentheses containing the target data type. Casting operator = ( ) or (data-type).

Example 1,

```
int num1 = 10;
float num2 = 5.5;
float result = num1 + (int)num2;
cout << result << endl; → Output = 15.00
```

Example 2,

```
double pi = 3.14159265;
int pipi = (int) pi;
cout << pipi << endl; → output = 3
```

Example 3,

```
float f = 65.35;
char ch = (char) f ;
cout << ch << endl; → output = A
```

Example 4,

```
int a=10;
int b = 3.0; float c= a/b;
cout << c << endl; → Output= 3
```

NOTE –

```
int / int = int
float / int = float
int / float = float
```

__/__/____

* TIME & SPACE COMPLEXITY →

- **Time Complexity** → Amount of time taken by an algorithm to run as a function of length of input.

Function of length of input means that time complexity is directly proportional to the input value.

If we increase or decrease the input value, the time complexity will increase or decrease respectively.

$$f \propto N$$

Here, f refers to the function of time complexity or number of operations performed by the CPU and that is directly proportional to the input value.

In simpler words, we can say **time complexity is CPU time utilisation.**

The time in the time complexity is not the actual time, it is the CPU operations which are defined using mathematical quantities.

= Why to study?

→ To reduce CPU operations.
→ Resources are limited.
→ To find optimal and efficient solution.

Suppose, there is

Algo A	Algo B
prints 1→N	prints 1→N
It takes CPU high processing	It takes CPU low processing.

So, Algo B will be an optimal solution.

- **Space complexity** → Amount of space taken by an algorithm to run as a function of length of input.

If we increase or decrease the input value, the space complexity will increase or decrease respectively.

Suppose,

```
int a = 1; // variable
int b[5]; // array
int n; cin>>n; for (int i=0; i<n ; i++)
{
}
```

So, when we increase or decrease the value of n, the variable a and array b remains the same or we can say that they will take time & space complexity equal as it is in the start. So, its time & space complexity is O(1) i.e. constant because on changing the value of n, there is no change in variable and array.

In Contrast,

```
int n;
cin>>n;
int * b = new int[n]; //dynamically allocated
//Print array b
  for (int i=0; i<n;i++){
        cout<< b[i];
  }
```

Suppose, $n=2$ → b[0], b[1] , $n=4$ → b[0], b[1], b[2], b[3]

So, here space complexity is changing while changing the value of n.
Space complexity would be linear here i.e. O(n).

* **UNIT TO REPRESENT COMPLEXITY →**

As everything is measured in some units, complexity also have some units to measure it.

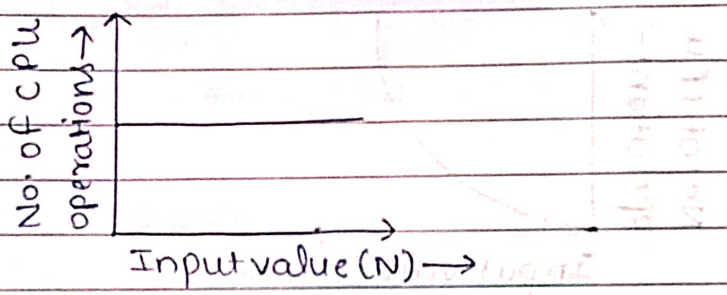
1. Big O : Upper Bound (Worst-to-worst case)
2. Theta θ : Average case
3. Omega Ω: lower Bound (Best case)

Mostly, Complexity is represented using Big O.

* Big O : Complexities

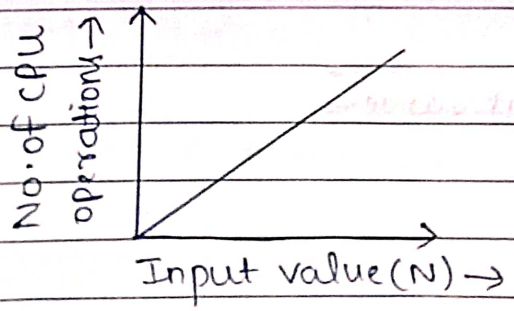
1. Constant time : O(1)
2. linear time : O(n)
3. logarithmic time : O(log N)
4. Quadratic time : O(N^2)
5. Cubic time : O(N^3)

Constant time Graph →



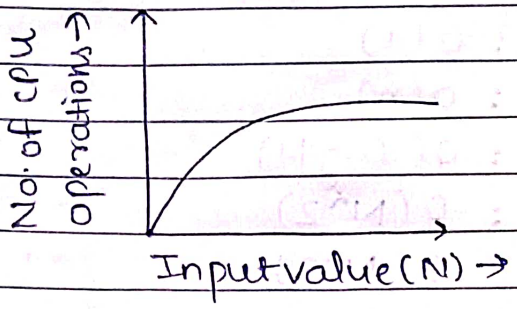
On increasing the value of N, there is no effect on no. of CPU operations.

linear time Graph →

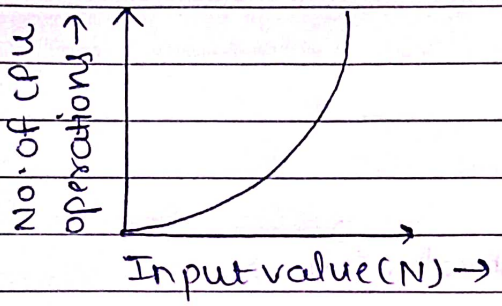


Number of CPU operations is directly proportional to the input value.

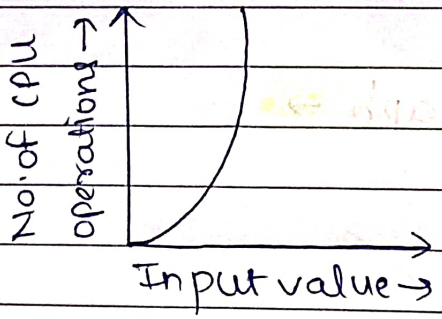
Logarithmic time graph →



Quadratic time graph →



Cubic time graph →



NOTE - for Nested loops, we multiply →

for instance,
```
for (int i=0; i<n; i++){
    for (int J=0; J<n; J++){
    }
}
```
Complexity will be $n \times n = n^2 \Rightarrow O(n^2)$

for two or more different loops, we add →

for instance,
```
for (int i=0; i<n; i++){
};
    for (int J=0; J<n; J++){
    }};
```
Complexity will be $n + n = 2n \Rightarrow O(n)$

Examples →

1. $f(n) = 2n^2 + 3n = O(2n^2) \Rightarrow O(n^2)$

Big O always looks for upper Bound.

2. $f(n) = 4n^4 + 3n^3 = O(4n^4) \Rightarrow O(n^4)$

3. $f(n) = n^2 + \log N \Rightarrow O(n^2)$

4. $f(n) = 200 \Rightarrow O(1)$

5. $f(n) = \frac{N}{4} \Rightarrow O(N)$

__/__/____

least to most complexities

$O(1)$
$O(\log N)$
$O(\sqrt{N})$
$O(N)$
$O(n \log n)$
$O(n^2)$
$O(n^3)$
$O(2^n)$
$O(n!)$
$O(n^n)$